Python Code

Your Name

December 20, 2024

1 Task 1: Implementing Computational Graphs

Q1 :Implement the forward and backward passes for the computational graph f3 below.

A1 :

```
def f3(s1, s2, y):
"""
Computes the forward and backward pass through the computational graph f3
from the homework PDF.

A few clarifications about the graph:
- The input y is an integer with y == 1 or y == 2; you do not need to
  compute a gradient for this input.
- The division nodes compute p1 = e1 / d and p2 = e2 / d
- The choose(p1, p2, y) node returns p1 if y is 1, or p2 if y is 2.

Inputs:
- s1, s2: Python floats
- y: Python integer, either equal to 1 or 2

Returns a tuple of:
- L: Python scalar giving the output of the graph
- grads: A tuple (grad_s1, grad_s2) giving the derivative of the output L
with respect to the inputs s1 and s2.
"""
assert y == 1 or y == 2
# Forward pass: Compute loss
L = None
###########################################################################
# TODO: Implement the forward pass for the computational graph f3 shown   #
# in the homework description. Store the loss in the variable L.          #
###########################################################################
e1 = math.exp(s1)
e2 = math.exp(s2)
e11 = e1
e12 = e1
e21 = e2
e22 = e2
d = e11 + e21
d1 = d
d2 = d
p1 = e12 / d1
p2 = e22 / d2
p_plus = p1*(2-y) - p2*(1-y)
L = -1 * math.log(p_plus)

###########################################################################
#                              END OF YOUR CODE                           #
###########################################################################

# Backward pass: Compute gradients
grad_s1, grad_s2 = None, None
###########################################################################
# TODO: Implement the backward pass for the computational graph f3 shown  #
# in the homework description. Store the gradients for each input         #
# variable in the corresponding grad variagbles defined above. You do not #
# need to compute a gradient for the input y since it is an integer.      #
#                                                                         #
# HINT: You may need an if statement to backprop through the choose node  #
```

```
    ############################################################################
    grad_L = 1
    grad_p_plus = -1*(grad_L / p_plus)
    grad_p1 = grad_p_plus * (2-y)
    grad_p2 = grad_p_plus * (y-1)
    grad_e12 = grad_p1 / d1
    grad_e22 = grad_p2 / d2
    grad_d1 = grad_p1 * e12 * -1 * (1 / d1**2)
    grad_d2 = grad_p2 * e22 * -1 * (1 / d2**2)
    grad_d = grad_d1 + grad_d2
    grad_e11 = grad_d
    grad_e21 = grad_d
    grad_e1 = grad_e11 + grad_e12
    grad_e2 = grad_e21 + grad_e22
    grad_s1 = math.exp(s1) * grad_e1
    grad_s2 = math.exp(s2) * grad_e2
    grad_y = grad_p_plus * (-p1 + p2)
    ############################################################################
    #                             END OF YOUR CODE                             #
    ############################################################################

    grads = (grad_s1, grad_s2)
    return L, grads
```

2 Task 2: Modular Backprop API

Q1 :Fully-connected layer: fc_forward and fc_backward.

A1 :

```
        def fc_forward(x, w, b):
    """
    Computes the forward pass for a fully-connected layer.

    The input x has shape (N, Din) and contains a minibatch of N
    examples, where each example x[i] has shape (Din,).

    Inputs:
    - x: A numpy array of shape (N, Din) giving input data
    - w: A numpy array of shape (Din, Dout) giving weights
    - b: A numpy array of shape (Dout,) giving biases

    Returns a tuple of:
    - out: output, of shape (N, Dout)
    - cache: (x, w, b)
    """
    out = None
    ############################################################################
    # TODO: Implement the forward pass. Store the result in out.               #
    ############################################################################
    out = x @ w + b
    ############################################################################
    #                             END OF YOUR CODE                             #
    ############################################################################
    cache = (x, w, b)
    return out, cache

```

```
        def fc_backward(grad_out, cache):
    """
    Computes the backward pass for a fully-connected layer.

    Inputs:
    - grad_out: Numpy array of shape (N, Dout) giving upstream gradients
    - cache: Tuple of:
      - x: A numpy array of shape (N, Din) giving input data
      - w: A numpy array of shape (Din, Dout) giving weights
      - b: A numpy array of shape (Dout,) giving biases

    Returns a tuple of downstream gradients:
    - grad_x: A numpy array of shape (N, Din) of gradient with respect to x
    - grad_w: A numpy array of shape (Din, Dout) of gradient with respect to w
```

```
    - grad_b: A numpy array of shape (Dout,) of gradient with respect to b
    """
    x, w, b = cache
    grad_x, grad_w, grad_b = None, None, None
    ###########################################################################
    # TODO: Implement the backward pass for the fully-connected layer         #
    ###########################################################################
    grad_x = grad_out @ w.T
    grad_w = x.T @ grad_out
    grad_b = np.sum(grad_out,axis=0)

    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return grad_x, grad_w, grad_b

```

Q2 :relu_forward and relu_backward

A2 :

```
        def relu_forward(x):
    """
    Computes the forward pass for the Rectified Linear Unit (ReLU) nonlinearity

    Input:
    - x: A numpy array of inputs, of any shape

    Returns a tuple of:
    - out: A numpy array of outputs, of the same shape as x
    - cache: x
    """
    out = None
    ###########################################################################
    # TODO: Implement the ReLU forward pass.                                  #
    ###########################################################################
    out = np.maximum(0,x)

    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    cache = x
    return out, cache

```

```
        def relu_backward(grad_out, cache):
    """
    Computes the backward pass for a Rectified Linear Unit (ReLU) nonlinearity

    Input:
    - grad_out: Upstream derivatives, of any shape
    - cache: Input x, of same shape as dout

    Returns:
    - grad_x: Gradient with respect to x
    """
    grad_x, x = None, cache
    ###########################################################################
    # TODO: Implement the ReLU backward pass.                                 #
    ###########################################################################
    grad_x = grad_out * (x > 0)
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return grad_x

```

Q3 :Softmax Loss Function:softmax_loss

A3 :

```python
        def softmax_loss(x, y):
    """
    Computes the loss and gradient for softmax (cross-entropy) loss function.

    Inputs:
    - x: Numpy array of shape (N, C) giving predicted class scores, where
      x[i, c] gives the predicted score for class c on input sample i
    - y: Numpy array of shape (N,) giving ground-truth labels, where
      y[i] = c means that input sample i has ground truth label c, where
      0 <= c < C.

    Returns a tuple of:
    - loss: Scalar giving the loss
    - grad_x: Numpy array of shape (N, C) giving the gradient of the loss with
      with respect to x
    """
    loss, grad_x = None, None
    ###########################################################################
    # TODO: Implement softmax loss                                            #
    ###########################################################################
    M = np.max(x,axis = 1,keepdims=True)
    x_minus_M = x - M
    exp_x = np.exp(x_minus_M)
    exp_x_each_row_sum = np.sum(exp_x,axis=1, keepdims=True)
    p_x = exp_x / exp_x_each_row_sum

    n = y.shape[0]

    loss = 0
    for i in range(n):
        true_label = y[i]
        p_true_label = p_x[i][true_label]
        log_p_true_label = np.log(p_true_label)
        loss += log_p_true_label
    loss = loss * -1 / n


    grad_x = p_x.copy()  # Make a copy of p_x
    grad_x[np.arange(n), y] -= 1
    grad_x /= n
    ###########################################################################
    #                              END OF YOUR CODE                           #
    ###########################################################################
    return loss, grad_x

```

Q4 :L2 Regularization: l2_regularization which implements the L2 regularization loss

A4 :

```python
        def l2_regularization(w, reg):
    """
    Computes loss and gradient for L2 regularization of a weight matrix:

    loss = (reg / 2) * sum_i w_i^2

    Where the sum ranges over all elements of w.

    Inputs:
    - w: Numpy array of any shape
    - reg: float giving the regularization strength

    Returns:
    """
    loss, grad_w = None, None
    ###########################################################################
    # TODO: Implement L2 regularization.                                      #
    ###########################################################################
    sum_Wi_squre = np.sum(w**2)
    loss = (reg / 2) * sum_Wi_squre
    grad_w = reg * w
    ###########################################################################
    #                              END OF YOUR CODE                           #
    ###########################################################################
```

```python
    return loss, grad_w


```

3 Task 3: Implement a Two-Layer Network

Q1 :Complete the implementation of the TwoLayerNet class. Your implementations for the forward and backward methods should use the modular forward and backward functions that you implemented in the previous task.

A1 :

```python
        class TwoLayerNet(Classifier):
    """
    A neural network with two layers, using a ReLU nonlinearity on its one
    hidden layer. That is, the architecture should be:

    input -> FC layer -> ReLU layer -> FC layer -> scores
    """
    def __init__(self, input_dim=3072, num_classes=10, hidden_dim=512,
                 weight_scale=1e-3):
        """
        Initialize a new two layer network.

        Inputs:
        - input_dim: The number of dimensions in the input.
        - num_classes: The number of classes over which to classify
        - hidden_dim: The size of the hidden layer
        - weight_scale: The weight matrices of the model will be initialized
          from a Gaussian distribution with standard deviation equal to
          weight_scale. The bias vectors of the model will always be
          initialized to zero.
        """
        ###########################################################################
        # TODO: Initialize the weights and biases of a two-layer network.         #
        ###########################################################################
        self.W1 = weight_scale * np.random.randn(input_dim, hidden_dim)
        self.b1 = np.zeros(hidden_dim)
        self.W2 = weight_scale * np.random.randn(hidden_dim, num_classes)
        self.b2 = np.zeros(num_classes)
        ###########################################################################
        #                            END OF YOUR CODE                             #
        ###########################################################################

    def parameters(self):
        params = None
        ###########################################################################
        # TODO: Build a dict of all learnable parameters of this model.           #
        ###########################################################################
        params = {
            'W1': self.W1,
            'b1': self.b1,
            'W2': self.W2,
            'b2': self.b2,
        }

        ###########################################################################
        #                            END OF YOUR CODE                             #
        ###########################################################################
        return params

    def forward(self, X):
        scores, cache = None, None
        ###########################################################################
        # TODO: Implement the forward pass to compute classification scores      #
        # for the input data X. Store into cache any data that will be needed #
        # during the backward pass.                                               #
        ###########################################################################
        W1 = self.W1
        W2 = self.W2
        b1 = self.b1
        b2 = self.b2

```

```
        #first fc layer
        fc1_out, fc1_cache = fc_forward(X,W1,b1)
        #apply the Relu layer
        relu_out, relu_cache = relu_forward(fc1_out)
        #second fc layer
        fc2_out, fc2_cache = fc_forward(relu_out,W2, b2)
        cache = (fc1_cache, relu_cache, fc2_cache)
        scores = fc2_out


        ###########################################################################
        #                             END OF YOUR CODE                            #
        ###########################################################################
        return scores, cache

    def backward(self, grad_scores, cache):
        grads = None
        ###########################################################################
        # TODO: Implement the backward pass to compute gradients for all          #
        # learnable parameters of the model, storing them in the grads dict       #
        # above. The grads dict should give gradients for all parameters in       #
        # the dict returned by model.parameters().                                #
        ###########################################################################
        W1 = self.W1
        W2 = self.W2
        b1 = self.b1
        b2 = self.b2

        fc1_cache, relu_cache, fc2_cache = cache
        grad_relu, grads_W2, grads_b2 = fc_backward(grad_scores, fc2_cache)

        grad_fc1 = relu_backward(grad_relu,relu_cache)

        trash, grads_W1, grads_b1 = fc_backward(grad_fc1,fc1_cache)
        grads = {
            'W1' : grads_W1,
            'b1' : grads_b1,
            'W2' : grads_W2,
            'b2' : grads_b2,
        }


        ###########################################################################
        #                             END OF YOUR CODE                            #
        ###########################################################################
        return grads

```

4 Task4: Training Two-Layer Networks

Q1 : Implement the training_step function in the file neuralnettrainpẏ

A1 :

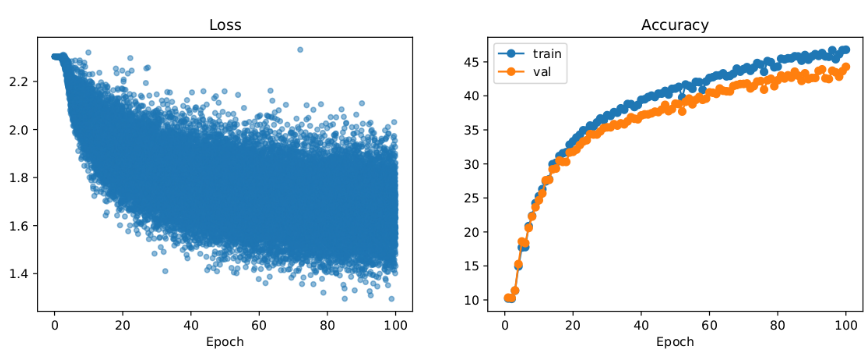
```
        def training_step(model, X_batch, y_batch, reg):
    """
    Compute the loss and gradients for a single training iteration of a model
    given a minibatch of data. The loss should be a sum of a cross-entropy loss
    between the model predictions and the ground-truth image labels, and
    an L2 regularization term on all weight matrices in the fully-connected
    layers of the model. You should not regularize the bias vectors.

    Inputs:
    - model: A Classifier instance
    - X_batch: A numpy array of shape (N, D) giving a minibatch of images
    - y_batch: A numpy array of shape (N,) where 0 <= y_batch[i] < C is the
      ground-truth label for the image X_batch[i]
    - reg: A float giving the strength of L2 regularization to use.

    Returns a tuple of:
    - loss: A float giving the loss (data loss + regularization loss) for the
      model on this minibatch of data
```

```
    - grads: A dictionary giving gradients of the loss with respect to the
      parameters of the model. In particular grads[k] should be the gradient
      of the loss with respect to model.parameters()[k].
    """
    loss, grads = None, None
    ###########################################################################
    # TODO: Compute the loss and gradient for one training iteration.         #
    ###########################################################################
    prediction, cache = model.forward(X_batch)

    #compute the data loss
    data_loss, grad_x = softmax_loss(prediction,y_batch)

    #compute the regularization loss
    reg_loss = 0
    for param_name, param in model.parameters().items():
        if param_name.startswith('W'):
            reg_loss += l2_regularization(param,reg)[0]
    #calculate the total loss
    loss = data_loss + reg_loss

    grads = model.backward(grad_x,cache)

    for param_name in grads:
        if param_name.startswith('W'):
            grads[param_name] += reg * model.parameters()[param_name]

    ###########################################################################
    #                              END OF YOUR CODE                           #
    ###########################################################################
    return loss, grads
```

Q2 :include the loss / accuracy plot for your best model, describe the hyperparameter settings you used, and give the final test-set performance of your model.

A2 :
the parameter I use here is:

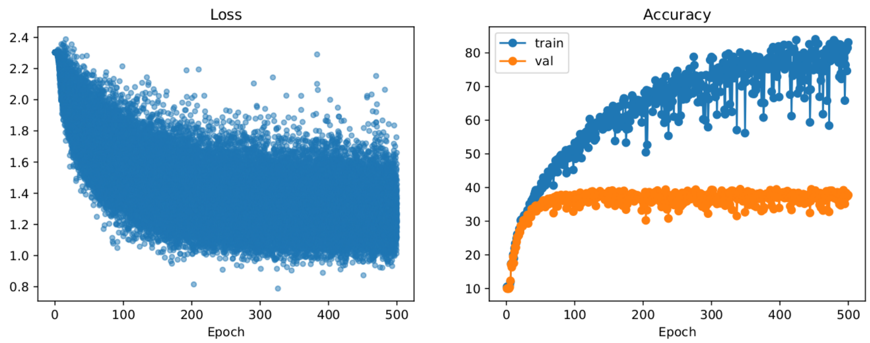
```
    # How much data to use for training
    num_train = 20000

    # Model architecture hyperparameters.
    hidden_dim = 64

    # Optimization hyperparameters.
    batch_size = 64
    num_epochs = 100
    learning_rate = 0.005
    reg = 0.01
```



Q3 : include the loss / accuracy plot for your overfit model and describe the hyperparameter settings you used.

A3 :
the parameter I use here is:

```
        # How much data to use for training
    num_train = 3000

    # Model architecture hyperparameters.
    hidden_dim = 64

    # Optimization hyperparameters.
    batch_size = 32
    num_epochs = 500
    learning_rate = 0.01
    reg = 0.01

```



5 Task 5: Train Your Own Classification Model

Q1 :Submit the notebook (with outputs) that trains with your best combination of model architecture, optimizer and training parameters, and evaluates on the test set to report an accuracy at the end.

A1 :

EECS442_HW4_task5_report

March 19, 2024

1 EECS 442 Homework 4: Fashion-MNIST Classification

In this part, you will implement and train Convolutional Neural Networks (ConvNets) in PyTorch to classify images. Unlike HW4 Secion 1, backpropagation is automatically inferred by PyTorch, so you only need to write code for the forward pass.

Before we start, please put your name and UMID in following format Firstname LASTNAME, #00000000 // e.g.) David FOUHEY, #12345678

Your Answer:
Hello EECS442 #12345678

1.1 Setup

[ ]:
```
# Run the command in the terminal if it failed on local Jupyter Notebook,␣
↪remove "!" before each line
!pip install torchsummary
```

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm # Displays a progress bar

import torch
from torch import nn
from torch import optim
import torch.nn.functional as F
from torchsummary import summary
from torchvision import datasets, transforms
from torch.utils.data import Dataset, Subset, DataLoader, random_split
```

[ ]:
```
if torch.cuda.is_available():
    print("Using the GPU. You are good to go!")
    device = 'cuda'
else:
    print("Using the CPU. Overall speed may be slowed down")
    device = 'cpu'
```

```
Using the CPU. Overall speed may be slowed down
```

1.2 Loading Dataset

The dataset we use is Fashion-MNIST dataset, which is available at https://github.com/zalandoresearch/fashion-mnist and in torchvision.datasets. Fashion-MNIST has 10 classes, 60000 training+validation images (we have splitted it to have 50000 training images and 10000 validation images, but you can change the numbers), and 10000 test images.

[ ]:
```
# Load the dataset and train, val, test splits
print("Loading datasets...")
# Transform from [0,255] uint8 to [0,1] float,
# then normalize to zero mean and unit variance
FASHION_transform = transforms.Compose([
                        transforms.ToTensor(),
                        transforms.Normalize([0.2859], [0.3530])
                    ])
FASHION_trainval = datasets.FashionMNIST('.', download=True, train=True,
                                         transform=FASHION_transform)
FASHION_train = Subset(FASHION_trainval, range(50000))
FASHION_val = Subset(FASHION_trainval, range(50000, 60000))
FASHION_test = datasets.FashionMNIST('.', download=True, train=False,
                                     transform=FASHION_transform)
print("Done!")
```

```
Loading datasets…
Done!
```

Now, we will create the dataloder for train, val and test dataset. You are free to experiment with different batch sizes.

[ ]:
```
# Create dataloaders
##############################################################################
# TODO: Experiment with different batch sizes                                #
##############################################################################
batch_size = 50
##############################################################################
#                              END OF YOUR CODE                              #
##############################################################################
trainloader = DataLoader(FASHION_train, batch_size=batch_size, shuffle=True)
valloader = DataLoader(FASHION_val, batch_size=batch_size, shuffle=True)
testloader = DataLoader(FASHION_test, batch_size=batch_size, shuffle=True)
```

1.3 Model

Initialize your model and experiment with with different optimizers, parameters (such as learning rate) and number of epochs.

[ ]:
```
class Network(nn.Module):
    def __init__(self):
        super().__init__()
```

```
    ␣
↪################################################################################
        # TODO: Design your own network, define layers here.                  ␣
↪        #
        # Here We provide a sample of two-layer fc network from HW4 Part3.     ␣
↪       #
        # Your solution, however, should contain convolutional layers.         ␣
↪       #
        # Refer to PyTorch documentations of torch.nn to pick your layers.     ␣
↪       #
        # (https://pytorch.org/docs/stable/nn.html)                            ␣
↪       #
        # Some common choices: Linear, Conv2d, ReLU, MaxPool2d, AvgPool2d,␣
↪Dropout   #
        # If you have many layers, use nn.Sequential() to simplify your code   ␣
↪       #
    ␣
↪################################################################################
        self.conv1 = nn.
↪Conv2d(in_channels=1,out_channels=16,kernel_size=3,stride=1,padding=1)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32,kernel_size=3,␣
↪stride=1, padding=1)

        self.relu = nn.ReLU()
        self.pool = nn.MaxPool2d(kernel_size=2,stride=2)

        self.fc1 = nn.Linear(32*7*7, 128)
        self.fc2 = nn.Linear(128, 10)


        ␣
↪################################################################################
        #                              END OF YOUR CODE                        ␣
↪       #
    ␣
↪################################################################################

    def forward(self, x):
    ␣
↪################################################################################
        # TODO: Design your own network, implement forward pass here           ␣
↪       #
    ␣
↪################################################################################
        x = self.pool(self.relu(self.conv1(x)))
        x = self.pool(self.relu(self.conv2(x)))
        x = x.view(-1,32*7*7)
```

```
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x
       ␣
 ↪###########################################################################
        #                              END OF YOUR CODE                        ␣
 ↪      #
       ␣
 ↪###########################################################################

model = Network().to(device)
criterion = nn.CrossEntropyLoss() # Specify the loss layer
print('Your network:')
print(summary(model, (1,28,28), device=device)) # visualize your model

##############################################################################
# TODO: Modify the lines below to experiment with different optimizers,      #
# parameters (such as learning rate) and number of epochs.                   #
##############################################################################
# Set up optimization hyperparameters
learning_rate, weight_decay, num_epoch = 0.001, 0.001, 10
optimizer = optim.Adam(model.parameters(), lr = learning_rate,␣
 ↪weight_decay=weight_decay)
##############################################################################
#                              END OF YOUR CODE                              #
##############################################################################
```

```
Your network:
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 28, 28]             160
              ReLU-2           [-1, 16, 28, 28]               0
         MaxPool2d-3           [-1, 16, 14, 14]               0
            Conv2d-4           [-1, 32, 14, 14]           4,640
              ReLU-5           [-1, 32, 14, 14]               0
         MaxPool2d-6             [-1, 32, 7, 7]               0
            Linear-7                  [-1, 128]         200,832
              ReLU-8                  [-1, 128]               0
            Linear-9                   [-1, 10]           1,290
================================================================
Total params: 206,922
Trainable params: 206,922
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.33
```

```
Params size (MB): 0.79
Estimated Total Size (MB): 1.12
----------------------------------------------------------------
None
```

Run the cell below to start your training, we expect you to achieve over **85%** on the test set. A valid solution that meet the requirement take no more than **10 minutes** on normal PC Intel core CPU setting. If your solution takes too long to train, try to simplify your model or reduce the number of epochs.

[ ]:
```
%%time
def train(model, trainloader, valloader, num_epoch=10):  # Train the model
    print("Start training...")
    trn_loss_hist = []
    trn_acc_hist = []
    val_acc_hist = []
    model.train()  # Set the model to training mode
    for i in range(num_epoch):
        running_loss = []
        print('-----------------Epoch = %d-----------------' % (i+1))
        for batch, label in tqdm(trainloader):
            batch = batch.to(device)
            label = label.to(device)
            optimizer.zero_grad()  # Clear gradients from the previous iteration
            # This will call Network.forward() that you implement
            pred = model(batch)
            loss = criterion(pred, label)  # Calculate the loss
            running_loss.append(loss.item())
            loss.backward()  # Backprop gradients to all tensors in the network
            optimizer.step()  # Update trainable weights
        print("\n Epoch {} loss:{}".format(i+1, np.mean(running_loss)))

        # Keep track of training loss, accuracy, and validation loss
        trn_loss_hist.append(np.mean(running_loss))
        trn_acc_hist.append(evaluate(model, trainloader))
        print("\n Evaluate on validation set...")
        val_acc_hist.append(evaluate(model, valloader))
    print("Done!")
    return trn_loss_hist, trn_acc_hist, val_acc_hist


def evaluate(model, loader):  # Evaluate accuracy on validation / test set
    model.eval()  # Set the model to evaluation mode
    correct = 0
    with torch.no_grad():  # Do not calculate grident to speed up computation
        for batch, label in tqdm(loader):
            batch = batch.to(device)
            label = label.to(device)
```

```python
            pred = model(batch)
            correct += (torch.argmax(pred, dim=1) == label).sum().item()
        acc = correct/len(loader.dataset)
        print("\n Evaluation accuracy: {}".format(acc))
        return acc


trn_loss_hist, trn_acc_hist, val_acc_hist = train(model, trainloader,
                                                  valloader, num_epoch)

##############################################################################
# TODO: Note down the evaluation accuracy on test set                        #
##############################################################################
print("\n Evaluate on test set")
evaluate(model, testloader)
```

```
Start training…
-----------------Epoch = 1-----------------

100%|     | 1000/1000 [00:11<00:00, 88.20it/s]


 Epoch 1 loss:0.45427982886135576

100%|     | 1000/1000 [00:08<00:00, 116.68it/s]


 Evaluation accuracy: 0.88322

 Evaluate on validation set…

100%|     | 200/200 [00:01<00:00, 112.52it/s]


 Evaluation accuracy: 0.8756
-----------------Epoch = 2-----------------

100%|     | 1000/1000 [00:14<00:00, 68.83it/s]


 Epoch 2 loss:0.3132331562563777

100%|     | 1000/1000 [00:13<00:00, 75.81it/s]


 Evaluation accuracy: 0.90188

 Evaluate on validation set…

100%|     | 200/200 [00:02<00:00, 76.01it/s]


 Evaluation accuracy: 0.8898
```

```
-----------------Epoch = 3-----------------
100%|     | 1000/1000 [00:20<00:00, 47.72it/s]

 Epoch 3 loss:0.27816830180585383
100%|     | 1000/1000 [00:20<00:00, 48.48it/s]

 Evaluation accuracy: 0.90692

 Evaluate on validation set…
100%|     | 200/200 [00:04<00:00, 48.26it/s]

 Evaluation accuracy: 0.8909
-----------------Epoch = 4-----------------
100%|     | 1000/1000 [00:32<00:00, 30.58it/s]

 Epoch 4 loss:0.25694305121526123
100%|     | 1000/1000 [00:23<00:00, 43.44it/s]

 Evaluation accuracy: 0.9169

 Evaluate on validation set…
100%|     | 200/200 [00:04<00:00, 42.62it/s]

 Evaluation accuracy: 0.8977
-----------------Epoch = 5-----------------
100%|     | 1000/1000 [00:33<00:00, 29.63it/s]

 Epoch 5 loss:0.24221466190367938
100%|     | 1000/1000 [00:26<00:00, 38.06it/s]

 Evaluation accuracy: 0.92516

 Evaluate on validation set…
100%|     | 200/200 [00:05<00:00, 35.77it/s]

 Evaluation accuracy: 0.906
-----------------Epoch = 6-----------------
100%|     | 1000/1000 [00:34<00:00, 28.83it/s]
```

```
 Epoch 6 loss:0.22939964060112833

100%|     | 1000/1000 [00:26<00:00, 37.96it/s]


 Evaluation accuracy: 0.92846

 Evaluate on validation set…

100%|     | 200/200 [00:04<00:00, 42.81it/s]


 Evaluation accuracy: 0.906
-----------------Epoch = 7-----------------

100%|     | 1000/1000 [00:35<00:00, 28.18it/s]


 Epoch 7 loss:0.21812415209040045

100%|     | 1000/1000 [00:26<00:00, 38.35it/s]


 Evaluation accuracy: 0.93176

 Evaluate on validation set…

100%|     | 200/200 [00:04<00:00, 43.88it/s]


 Evaluation accuracy: 0.9114
-----------------Epoch = 8-----------------

100%|     | 1000/1000 [00:33<00:00, 30.26it/s]


 Epoch 8 loss:0.21002072999626398

100%|     | 1000/1000 [00:24<00:00, 41.14it/s]


 Evaluation accuracy: 0.93376

 Evaluate on validation set…

100%|     | 200/200 [00:04<00:00, 41.02it/s]


 Evaluation accuracy: 0.9128
-----------------Epoch = 9-----------------

100%|     | 1000/1000 [00:35<00:00, 28.40it/s]


 Epoch 9 loss:0.20510010103322565
```

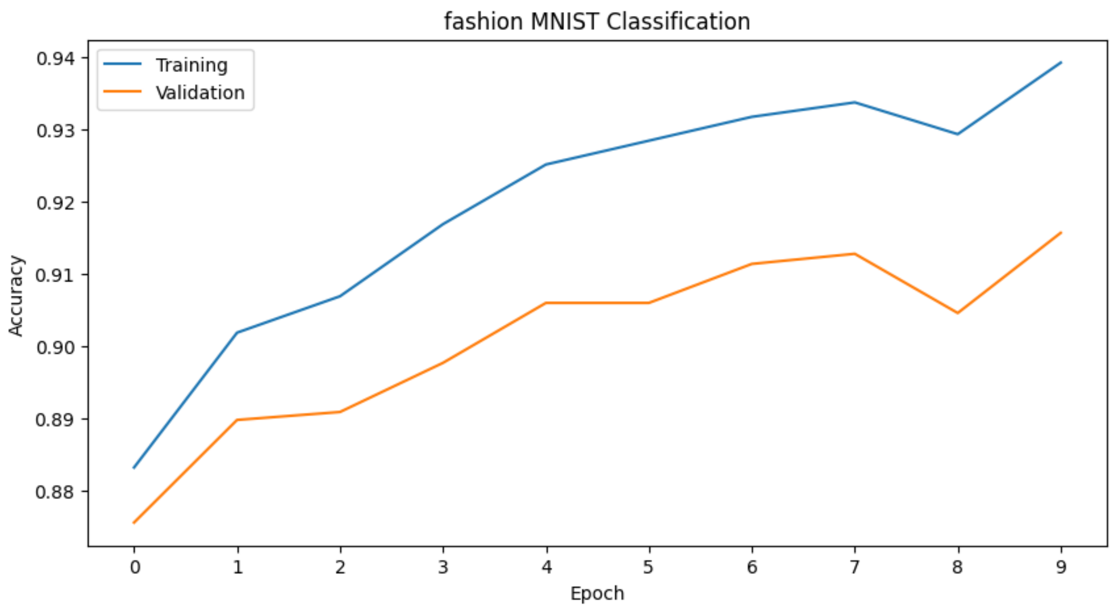
```
100%|      | 1000/1000 [00:25<00:00, 39.44it/s]


 Evaluation accuracy: 0.92936

 Evaluate on validation set…

100%|      | 200/200 [00:04<00:00, 41.56it/s]


 Evaluation accuracy: 0.9046
-----------------Epoch = 10-----------------

100%|      | 1000/1000 [00:36<00:00, 27.54it/s]


 Epoch 10 loss:0.19699719595909118

100%|      | 1000/1000 [00:22<00:00, 44.43it/s]


 Evaluation accuracy: 0.93926

 Evaluate on validation set…

100%|      | 200/200 [00:04<00:00, 43.24it/s]


 Evaluation accuracy: 0.9157
Done!

 Evaluate on test set

100%|      | 200/200 [00:04<00:00, 43.46it/s]


 Evaluation accuracy: 0.9112
CPU times: user 2h 1min 28s, sys: 2.4 s, total: 2h 1min 31s
Wall time: 9min 11s
```

[ ]:
```
0.9112
```

Once your training is complete, run the cell below to visualize the training and validation accuracies across iterations.

[ ]:
```
##############################################################################
# TODO: Submit the accuracy plot                                             #
##############################################################################
# visualize the training / validation accuracies
x = np.arange(num_epoch)
# train/val accuracies for MiniVGG
plt.figure()
```

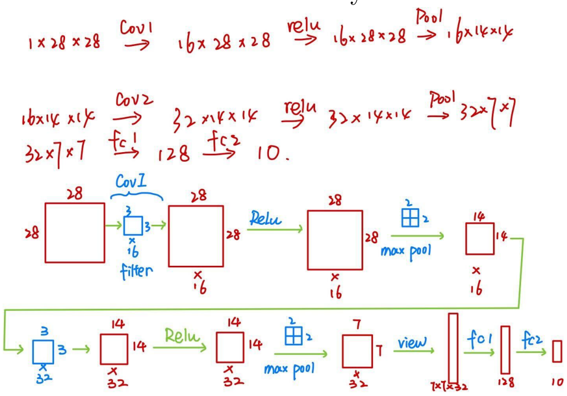
```
plt.plot(x, trn_acc_hist)
plt.plot(x, val_acc_hist)
plt.legend(['Training', 'Validation'])
plt.xticks(x)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('fashion MNIST Classification')
plt.gcf().set_size_inches(10, 5)
plt.savefig('part1.png', dpi=300)
plt.show()
```



Q2 :Report the detailed architecture of your best model. Include information on hyperparameters chosen for training and a plot showing both training and validation accuracy across iterations.

A2 :

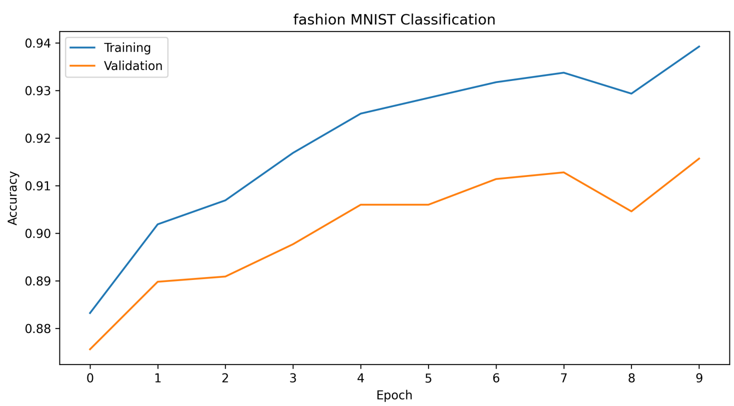
The detialed architecture of my best model:



The hyperparameters i chosen to build this cnn:

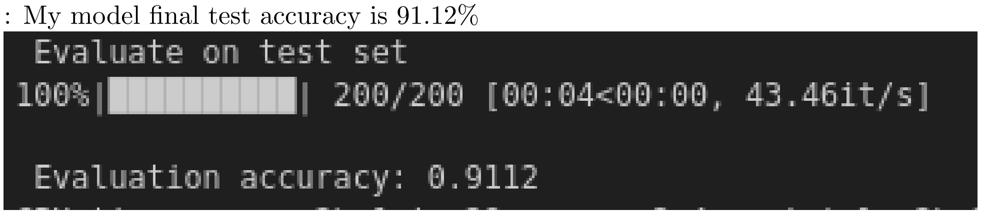
```
            #set up the batch_size
            batch_size = 50
            # Set up optimization hyperparameters
            learning_rate, weight_decay, num_epoch = 0.001, 0.001, 10
            optimizer = optim.Adam(model.parameters(), lr = learning_rate, weight_decay=weight_decay)

```

plot showing both training and validation accuracy:



Q3 : Report the accuracy of your best model on the test set.

A3 : My model final test accuracy is 91.12%



```
 Evaluate on test set
100%|██████████| 200/200 [00:04<00:00, 43.46it/s]

 Evaluation accuracy: 0.9112
```

6 Task 6: Pre-trained NN

Q1 :One image (img1) where the pretrained model gives reasonable predictions, and produces a category label that seems to correctly describe the content of the image

A1 :

Q2 : One image (img2) where the pretrained model gives unreasonable predictions, and produces a category label that does not correctly describe the content of the image.

A2 :